Written Assignment 3
Due Thursday, May 28, 2015 at 11:00am

This assignment asks you to prepare written answers to questions on type checking and operational semantics. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work. Assignments can be submitted electronically as a PDF by 11:00 AM PDT on Thursday, May 28 on scoryst.com.

1. Consider the following class definitions.

```java
class X {
    z: Bool;
    c: Int;
    s: String;
    x: SELF_TYPE;
    foo(): SELF_TYPE { x };
    getC(): Int { c };
};
class Y inherits X {
    y: SELF_TYPE;
    h(b: Object): Object { (* EXPRESSION *) };
};
```

Assume that the type checker implements the rules described in the lectures and in the Cool Reference Manual. For each of the following expressions, occurring in place of (* EXPRESSION *) in the body of the method `h`, show the static type inferred by the type checker for the expression. If the expression causes a type error, give a brief explanation of why the appropriate type checking rule for the expression cannot be applied.

(a) `x <- y.foo()`  
(b) `s.length() + z`  
(c) `case x.foo() of`  
    `z: X => z.getG();`  
    `z: Y => z;`  
    `l: Object => true;`  
    `esac`

2. A type derivation shows the inductive proof of a typing judgement as a tree. For example, the type derivation for \( O[\text{Int}/x] \vdash x + 1 : \text{Int} \) is given as follows:

\[
\begin{align*}
\text{O[\text{Int}/x](x) = Int} & \\
\text{\hline} & \\
\text{O[\text{Int}/x] \vdash x : \text{Int}} & \\
\text{O[\text{Int}/x] \vdash 1 : \text{Int}} & \\
\text{\hline} & \\
\text{O[\text{Int}/x] \vdash x + 1 : \text{Int}} & 
\end{align*}
\]

Show the type derivation for the following typing judgement:
3. Suppose we want to add a special type to Cool that can either be an `Int` or a special value that represents “no result.” Our type `MaybeInt` will take two forms, `Some(n)`, where `n` is an integer, and `Nothing`. Suppose we can create values of this form from the following compiler-implemented methods: (assume that they are defined `Object` so they are always accessible):

```cpp
createSomething(n : Int) : MaybeInt
createNothing() : MaybeInt
```

For example, you could use `MaybeInt` to write a safe division function that handles division by zero:

```cpp
safeDivision(numerator : Int, denominator : Int) : MaybeInt {
    if (denominator == 0) then
        createNothing()
    else
        createSomething(numerator / denominator)
    fi
}
```

Another example of a function that uses `MaybeInt` would be an integer parsing function. This function will return a `MaybeInt` with value `Some(n)` if the parse succeeds and value `Nothing` if it fails. The signature would then be:

```cpp
parseInt(str: String) : MaybeInt
```

As an example, `parseInt("123")` would return `Some(123)` and `parseInt("124a")` would return `Nothing`. To be able to use the value inside of a `MaybeInt`, we will introduce pattern matching syntax to capture the value and bind it to a value in an expression. Similar to how a `switch` statement in other languages goes to a branch depending on the value of a variable, a `match` statement will go to a branch depending on the form of `MaybeInt`, while also possibly introducing a new value into the scope. The value of a `match` expression is then the value of the expression on the right side of the branch that was taken.

```cpp
let maybe_int : MaybeInt <- parseInt("23") in
    match(maybe_int) {
    Some(n) => "The number is ".concat(n.to_string()),
    Nothing => "There was no number"
    }
```

This expression has the value “The number is 23”. The explicit grammar rule for `match` is:

```
expr ::= match (expr) { Some(ID) => expr, Nothing => expr }
```

The identifier in the `Some` branch is introduced into the scope of the expression on the right side and initialized to the value inside the `MaybeInt`. Your task is to write a type checking rule for `match` `{ Some(id) => e2, Nothing => e3 }`, as well as the operational semantics. Hint: For the operational semantics, model your solution off of the operational semantics for `Let` and `If-True/False`. Assume that when looking up a `MaybeInt` in the store that it will either return `Some(v)` or `Nothing`, so you may need to write two separate rules.